jnprsr – A parser for Juniper configuration files

Markus Jungbluth (markusju)

2025-02-12

DENOG Meetup 2025-1

Disclaimer

Opinions and views expressed are solely my own and do not express the views or opinions of my employer. The software presented herein is neither supported nor endorsed in any way by Juniper Networks Inc.

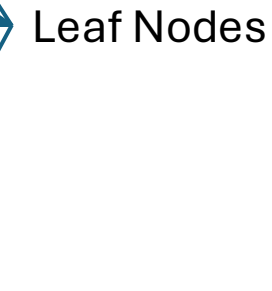
Why? / Motivation

- In 2019 I was building a system for Juniper configuration templates
- I was very annoyed by the fact, that I could not easily compose configuration fragments into a single configuration
- I ended up using the Jinja2 “include“ feature but was not happy about it.
  - Indentation was messed up
  - I need to pay attention that I place the include at the right position
- I thought that there should be an easier way to merge Juniper configuration off-box.

➢ **A new weekend coding project was born!**

What does Juniper Configuration look like?

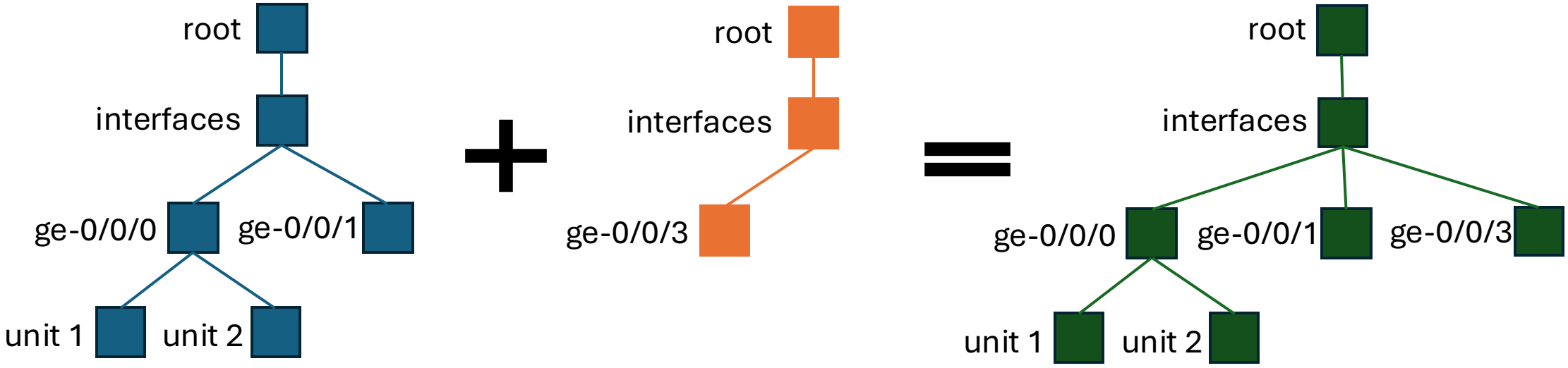
```
system {
    host-name ABC;
    domain-search [ google.com ];
}
interfaces {
    ge-0/0/0 {
        description "blah";
        unit 0 {
            family inet {
                address 1.2.3.5/24;
            }
        }
    }
    ge-0/0/1 {
        unit 0 {
            family inet {
                address 2.2.3.4/24;
            }
        }
    }
}
```



- Juniper configuration uses a hierarchical tree-like data structure.
- Different levels are expressed using curly-braces
- Consists of Container Nodes and Leaf Nodes
- Very similar to concepts found in programming languages like C, C++, Java

How do I approach this?

- Pretty soon, I realized that this was not going to be trivial.
- I would need to generate some kind of tree data structure from the configuration to be able to merge two configurations:



Wait a minute..

- Tree data structures..
- Curly-braces..
- Feels a bit like C, C++ or Java…

➢ **That sounds like a job for a parser!**

➢ **Juniper Configuration can be defined as a formal language!**

What is a formal language?

Consists of

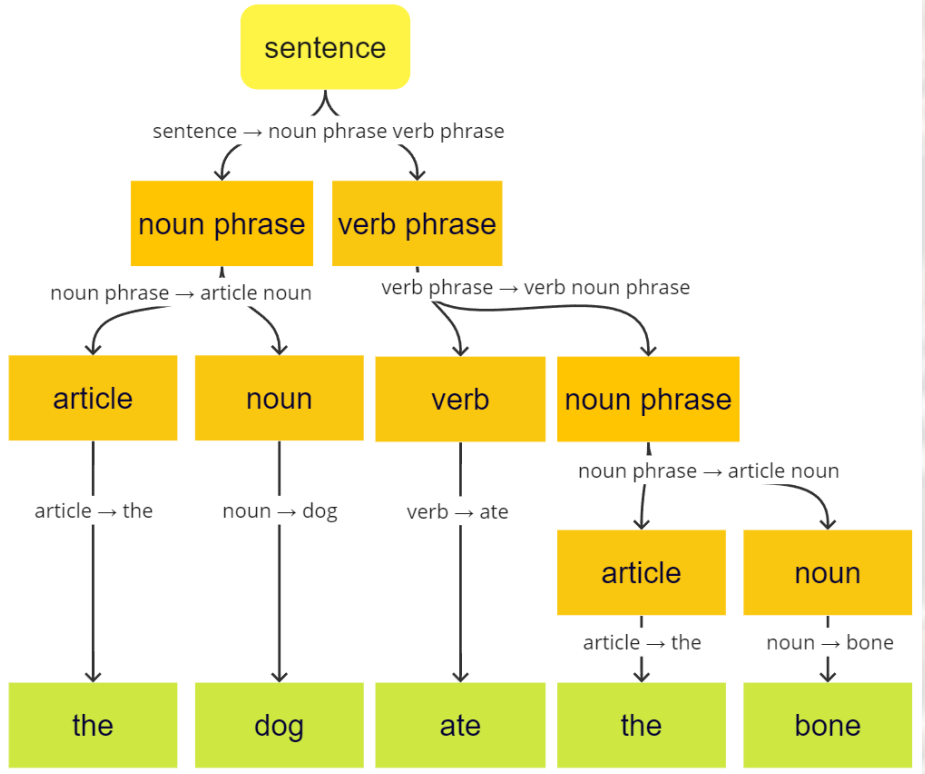
- **Alphabet** as a set of symbols
  {e, o, h, t, b, n, d, g, a}
- **Words** are strings whose letters/symbols are taken from an **Alphabet** according to a set of rules
- **Formal Grammar** is a set of (production) rules describing the syntax of the formal language

Non-Terminal Symbols = {sentence, noun, verb, article, noun phrase, verb phrase}

Terminal Symbols = {dog, bone, the, ate}

Production Rules = {sentence → noun phrase verb phrase,
noun phrase → article noun,
verb phrase → verb noun phrase,
noun → dog or bone,
verb → ate,
article → the}

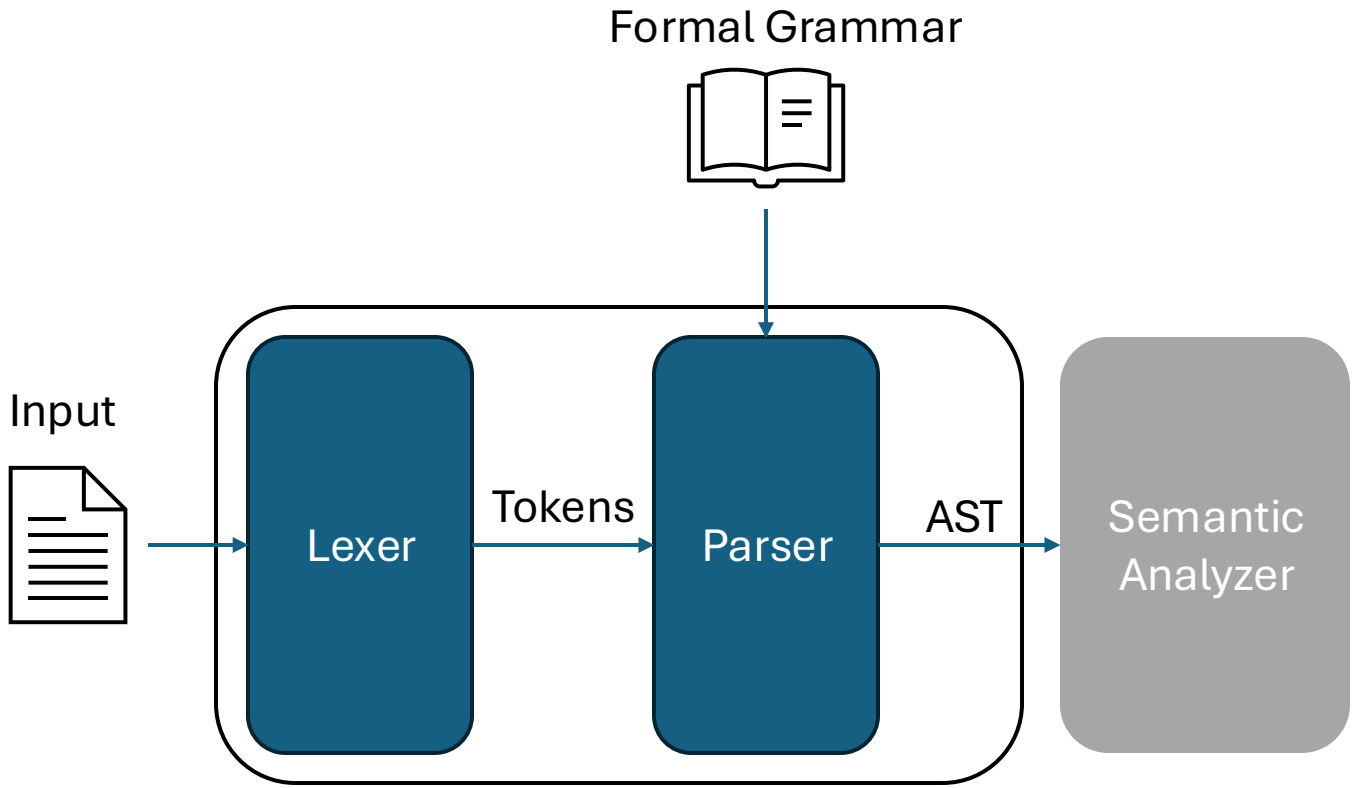
Start Symbol = sentence



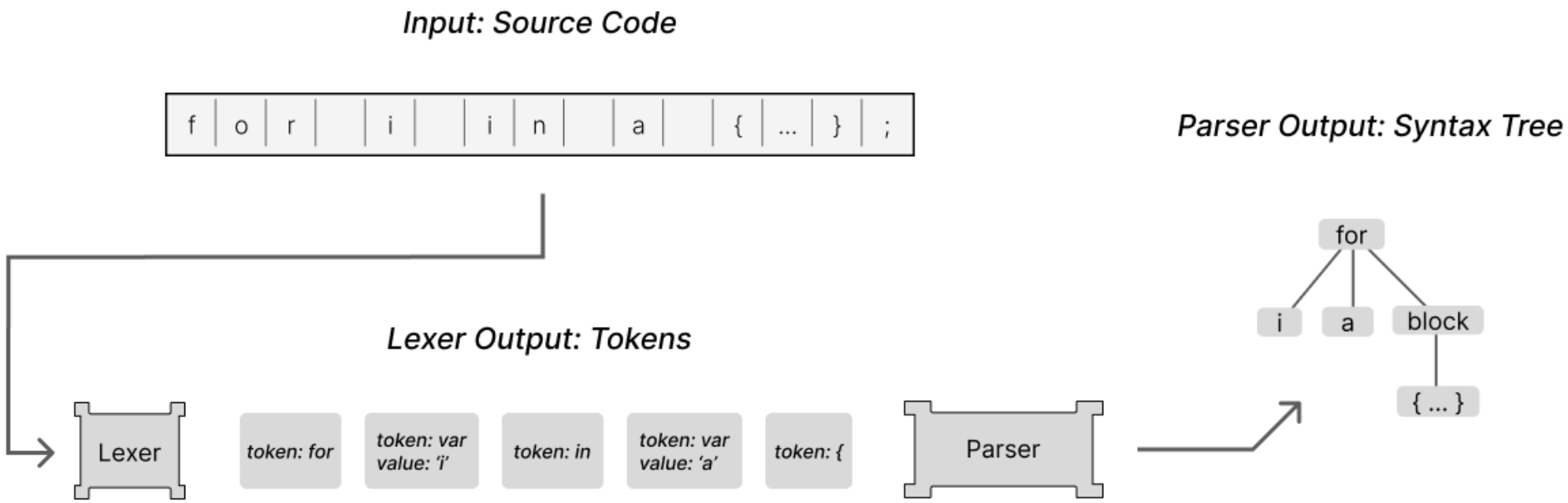
* I am not a theoretical computer scientist and have just shamelessly taken this from Wikipedia :)
Source: https://en.wikipedia.org/wiki/Formal_grammar

What is a parser?

- A parser is a piece of software that analyzes a given series of symbols and produces an Abstract Syntax Tree (AST)
- It is usually combined with a **Lexer** that converts a series of input characters into **tokens**.
- This **stream of tokens** is then read by the **parser** according to production rules defined by a **formal grammar**
- The **parser** then produces an **Abstract Syntax Tree**, which is then usually fed into a semantic analyzer



Parser Example



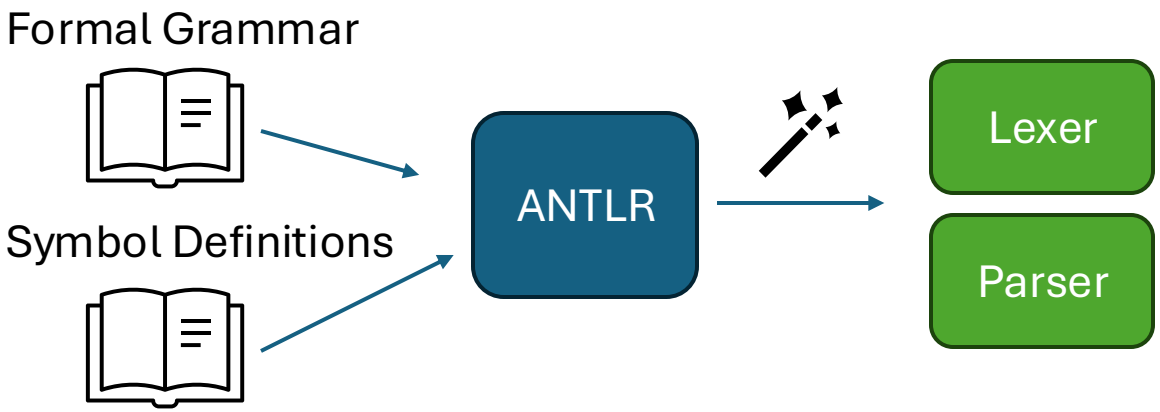
Source: https://www.beau-carlborg.com/blog/parsers-and-encoders-everywhere

Let‘s build a Juniper Config Parser!

- What do we need?
  - A definition of symbols
  - A formal grammar
  - A Lexer based on the symbol definitions
  - A Parser based on the formal grammar
  - Some brains to implement this!

There should be an easier way!

- **And there is**: ANTLR (**AN**other **T**ool for **L**anguage **R**ecognition) is a Parser Generator
- You supply **ANTLR** with the **formal grammar** and **symbol definitions** and it will generate an implementation of the Lexer and Parser in a programming language of your choice:
  - C++, C#, Dart, Java, JavaScript, PHP, Python3, Swift, TypeScript, Go



Let‘s build the parser

- I have based my work on existing ANTLR defintions created by the **batfish** project.
- Luckily I could almost use this as is and had a pretty solid basis for the parser.

Parser

```
parser grammar JuniperParser;

options {
  tokenVocab = JuniperLexer;
}

braced_clause
:
  OPEN_BRACE statement* CLOSE_BRACE
;

bracketed_clause
:
  OPEN_BRACKET word+ CLOSE_BRACKET
;

juniper_configuration
:
  statement+ EOF
;

statement
:
  (
    INACTIVE
    | REPLACE
  )? words += word+
  (
    braced_clause
    |
    (
      bracketed_clause terminator
    )
    | terminator
  )
;

terminator
:
  SEMICOLON
;

word
:
  WORD
;
```

Lexer

```
lexer grammar JuniperLexer;

options {
}

[…]

OPEN_BRACE
:
   '{'
;

OPEN_BRACKET
:
   '['
;

OPEN_PAREN
:
   '('
;

SEMICOLON
:
   ';'
;

WORD
:
   F_QuotedString
   | F_ParenString
   | F_WordChar+
;


[…]
```

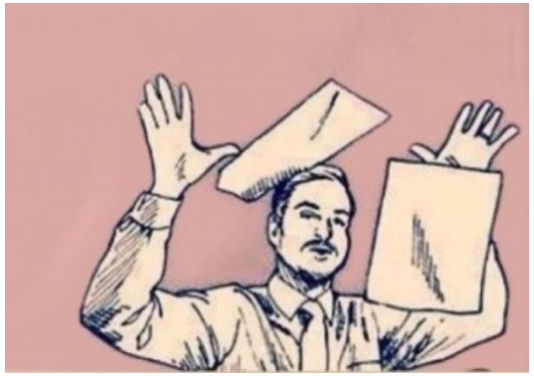
**shortened*

What‘s next?

- Now, I am able to generate an AST from a given Juniper configuration.
- First tests had shown that this worked pretty reliably!

➢ However, the AST tree data type produced by ANTLR is not really suitable for further use

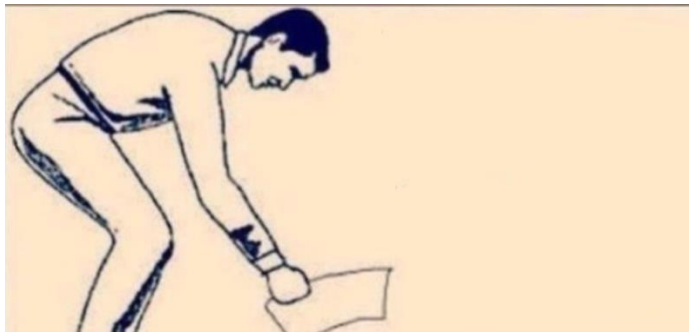
- ➢ No native way to make modifications to it, besides monkey patching
- ➢ No good helper functions to deal with the tree data structure (tree traversal, modification, etc)
- ➢ Not suitable for my goal of merging two configs! :(
- ➢ Project was tossed in the corner for a while.



Homer image source: https://www.deviantart.com/captainedwardteague/art/Home-Simpson-Woo-Hoo-837471740

How do we solve this?

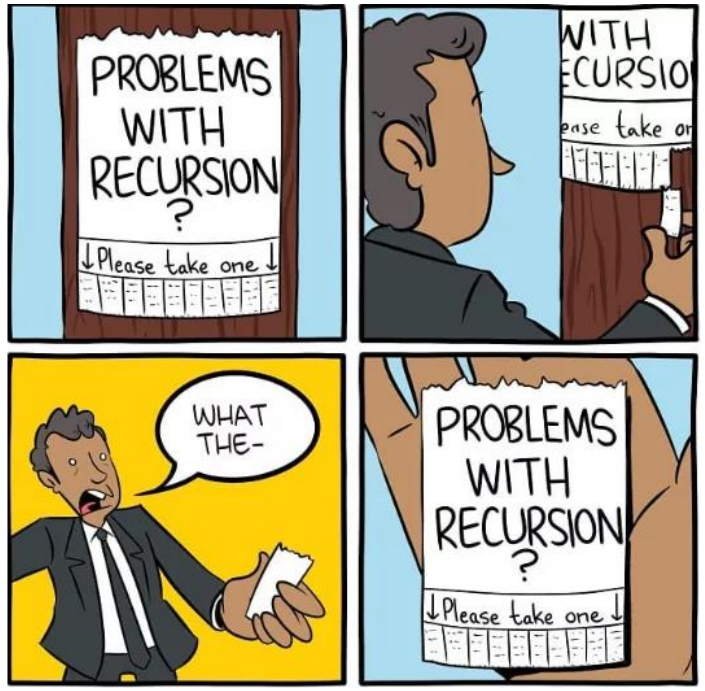
- ANTLR offers something called **ParseTreeListener** and **ParseTreeWalker**
- A **ParseTreeListener** defines various methods that are called when a specific part of the configuration is visited. For components, when a “statement“ is seen.
- A **ParseTreeWalker** allows to traverse a given AST and connect a **ParseTreeLisetner**.
- Using these components, I was able to easily convert the tree into another data format!
- I have chosen the **anytree library**, as it offers many helper functions around the tree data structure

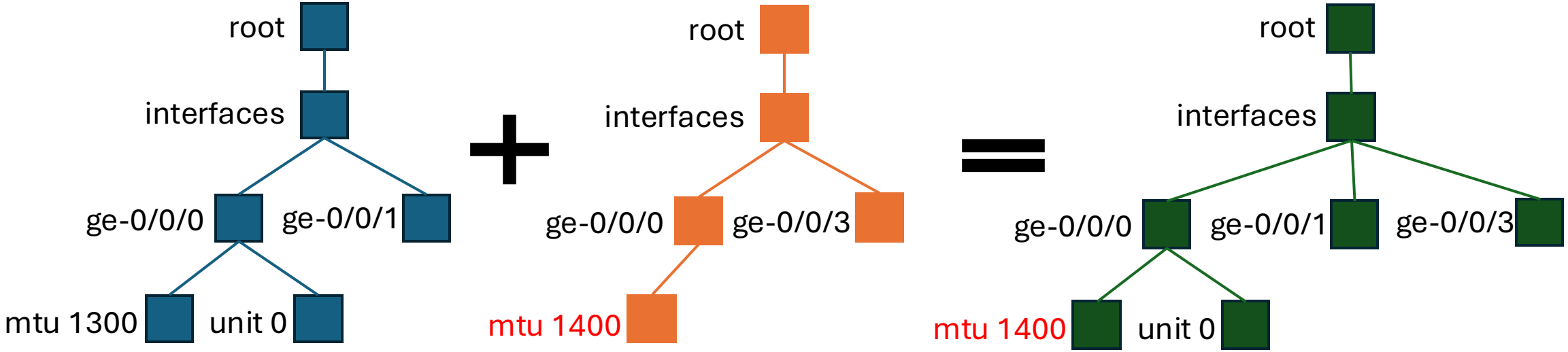


```python
class JuniperAST(JuniperParserListener):
    """
    This Class inherits from the JuniperParserListener created by ANTLR4.
    We are hooking onto various methods called during tree traversal to print out the
config.
    A buffer used inside the class collects the configuration tree reconstructed from
the the tree.
    """
    def __init__(self):
        self.root = JuniperASTNode("root")
        self.point = self.root
        self.depth = 0
        self.in_bracketed_clause = False

    def enterStatement(self, ctx: JuniperParser.StatementContext):
        node = JuniperASTNode(name="", parent=self.point)
        self.point = node
        # Exit a parse tree produced by JuniperParser#statement.

    def exitTerminator(self, ctx: JuniperParser.TerminatorContext):
        self.point = self.point.parent
        # Enter a parse tree produced by JuniperParser#word.

    def enterWord(self, ctx: JuniperParser.WordContext):
        if self.in_bracketed_clause:
            node = JuniperASTNode(name="", parent=self.point)
            self.point = node

        if len(self.point.name) < 1:
            self.point.name += ctx.getText()
            self.point.key = ctx.getText()
        else:
            self.point.name += " " + ctx.getText()
            self.point.value += " " + ctx.getText()

        if self.in_bracketed_clause:
            self.point = self.point.parent
        # Exit a parse tree produced by JuniperParser#word.

    def exitWord(self, ctx: JuniperParser.WordContext):
        pass
```

How are two trees merged?

- We need to define the target and source configuration.
  - In case a leaf node is overwritten by the merge operation, the value from the source will always overwrite the value of the target.
- What do we need to do to merge two trees?
  - We need to traverse through both tress at the same time and find common nodes and different nodes at each level.
    - In case a common node is found and it is a container node, we need to descend.
    - In case different nodes are found, they need to be added to the target tree.
- We need to apply this logic to every level of both trees: **RECURSION**





Target Configuration **Source Configuration** **Merged Configuration**

Let‘s merge then!

- Unfortunately, there are exceptions for configuration merging! :)
- Example 1

```
protocols {
    isis {
        interface lo0.0;
        interface ge-0/1/3.10;
    }
}
```

Target Configuration

+

```
protocols {
    isis {
        interface ge-0/1/4.10;
    }
}
```

Source Configuration

=

```
protocols {
    isis {
        interface lo0.0;
        interface ge-0/1/3.10;
        interface ge-0/1/4.10;
    }
}
```

In this case we have a leaf node with additional parameters. In this instance the interface leaf node will not be replaced, but added to the isis context.

Let‘s merge then!

- Unfortunately, there are exceptions for configuration merging! :)
- Example 2

```
system {
    host-name "blah";
}
```

+

```
system {
    host-name "abc";
}
```

Target Configuration

=

```
system {
    host-name "abc";
}
```

Source Configuration

In this case we also have a leaf node with additional parameters. In this instance however the leaf node will be replaced and not added as an additional node to the system context!

End of the story

- Merging is now implemented, and I have tried to capture all exceptions in the process.
  - Currently trying to cover as many as possible with test cases
- Software now delivers merge operations that are sufficient for my initial use case.

➢ **jnprsr was published**

What can you do with jnprsr?

- Can be used either directly **on your shell** or as **part of your own python script**
- On the Shell we currently support
  - Pretty-Printing
  - Sub-Tree Selection / Interactive CLI
  - Merge

On your shell: What can you do with jnprsr?

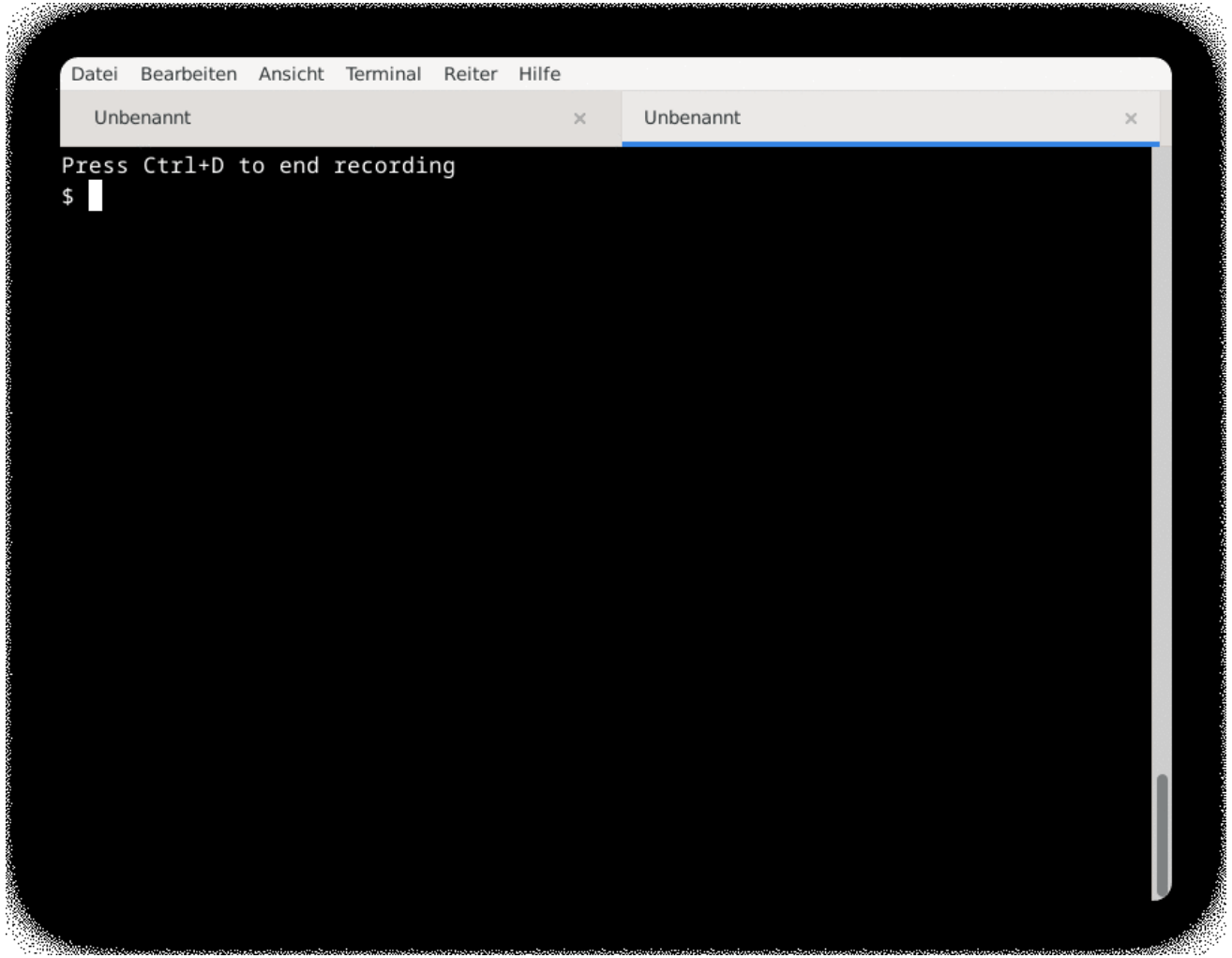
- Pretty-Printing

```
$ jnprsr-pretty
[Type CTRL+D or '!END' at a new line to end input]
interfaces { et-0/0/0 {description "More Bandwidth!"; unit 0 {family inet{address
192.0.2.1/24;}}}}
interfaces {
    et-0/0/0 {
        description "More Bandwidth!";
        unit 0 {
            family inet {
                address 192.0.2.1/24;
            }
        }
    }
}
```

```
$ cat what-a-mess.txt | jnprsr-pretty -s
interfaces {
    et-0/0/0 {
        description "More Bandwidth!";
        unit 0 {
            family inet {
                address 192.0.2.1/24;
            }
        }
    }
}
```

On your shell: What can you do with jnprsr?

- Sub-Tree Selection / Interactive CLI



On your shell: What can you do with jnprsr?

- Merge

```
$ cat test1.txt
system {
    host-name "test";
}
$ cat test2.txt
system {
    name-server {
        10.4.3.222;
    }
}
$ jnprsr-merge test1.txt test2.txt
system {
    host-name "test";
    name-server {
        10.4.3.222;
    }
}
```

In your script: What can you do with jnprsr?

```
import jnprsr

config = "system { host-name "test"; }"
ast = jnprsr.get_ast(config)
```

- You will almost always start with the **get_ast()** function. It will parse a given string and return an object of the type **JuniperASTNode**.
- This datatype inherits properties from the **anytree NodeMixin** type. So, you can work with this object the same way you would work with an anytree tree.

For ease of use we have created some Juniper Configuration specific helper functions:

- **render_config_from_ast(ast: JuniperASTNode) -> str**
  - This will return a textual representation of the AST, meaning it will converted back into configuration text.
- **render_ascii_tree_from_ast(ast: JuniperASTNode) -> str**
  - This will return an ascii tree representing the AST, because why not? :)
- **render_dict_from_ast(ast: JuniperASTNode) -> dict**
  - This will return a dictionary from a given configuration file.
- **merge(ast1: JuniperASTNode, ast2: JuniperASTNode) -> JuniperASTNode**
  - This will merge ast2 onto ast1
- **get_sub_tree(ast: JuniperASTNode, path: str) -> JuniperASTNode**
  - This will return the subtree at specified path, for example 'interfaces et-0/0/0'

What you can build with jnprsr

- Config Template Management Systems
- Configuration Validation
  - Is something correctly configured?
  - Are best-practices used?
- Comparing Configurations
  - Building a diff should be fairly easy
- Renderer that will output config in „set“ format
- Configuration Converter
  - Convert an existing Juniper config to the configuration of another vendor
  - Probably quite complex

Where can I get jnprsr?

- Code is available on GitHub
  - https://github.com/markusju/jnprsr
- You can install **jnprsr** using pip:

```
$ pip install jnprsr
```

- Do you want to work on this?
  See something I messed up?
  Want to add a new feature?

➢**Send me a pull request!**

Thank you!